Image Access Services Specification

Central Imagery Office
United States Imagery System

CIO Document No. CIO-2068

Release Date: 20 December, 1996
Version 1.1b
DRAFT

Acknowledgments

Many individuals and organizations provided support and technical contributions to this work, and acknowledge their participation here:

Dave Barnum (GTE)
Dwight Brown (NPIC/NEL)
Ron Burns (CIO)
Jeff Bushmire (Booz•Allen Hamilton)
Keith Butters (Rome Laboratories)
John Carney (Kodak)
Joe Coco (TASC)
Tim Daniel (CIO)
Bill Dowling (MITRE)
Chris Deschenes (TASC)
John Files (LMMS)
C. Scott Foshee (Booz•Allen Hamilton)
Rich Garrison (LMMS)
Leslie Gelman (SES)
Charlie Green (Sierra Concepts, Inc.)
Andy Hall (Rome Laboratories)
Ray Harrington (LMMS)
Tom Herron (MITRE)
Peggy Hwu (Booz•Allen Hamilton)
Don Joder (Booz•Allen Hamilton)
Mark Krug (SAIC)
Ros Lewis (Aerospace)
Dave Lutz (MITRE)
Raphael Malveau (MITRE)
John Marsh (MITRE)
Greg McBroome (SAIC)
Jim Meck (Booz•Allen Hamilton)
Tom Moore (GTE)
Tom Mowbray (MITRE)
Jack Needham (Harris)
Rick Nehrboss (Booz•Allen Hamilton)
Bill Nell (LMMS)
Rollie Olson (Loral)
Don Panzenhagen (Booz•Allen Hamilton)
Paul Parowski (Booz•Allen Hamilton)
Kathleen Perez-Lopez (Hughes)
Ed Rose (TRW)
Kathy Saint (Hughes)
John Salerno (Rome Laboratories)
Paul Silvey (MITRE)
Glen Speckert (TASC)
Noah Spivak (Booz•Allen Hamilton)
Shel Sutton (MITRE)
Alexis Thurman (Booz•Allen Hamilton)
John Tisaranni (MITRE)
Dave Weight (LMMS)
Jocelyn Yamamoto (TRW)
Ron Zahavi (MITRE)

Revision History

- Image Access Facility, Version 0.1 Straw 23 May 1995.
- Image Access Facility, Version 0.2 Tin 11 June 1995.
- Image Access Facility, Version 0.3 Aluminum 19 June 1995.
- Image Access Facility, Version 0.4 Copper - For USIS release June 21, 1995.
- Image Access Facility, Version 0.5 Nickel - Preliminary draft release for Image Access Working Group (IAWG) June 29, 1995.
- Image Access Facility, Version 0.6 Iron - This release will contain a relatively complete description of semantics and sequencing for sample implementation prototypers. July 12, 1995.
- Image Access Facility, Version 0.7 Silver - This release addresses comments received. September 6, 1995.
- Image Access Facility and Catalog Access Facility, Version 0.8 Gold - This release contains extensions based upon the additional architecture mining. February 8, 1996.
- Image Access Facility and Catalog Access Facility, Version 0.85 Gold Interim - Update for release and comment on March 22, 1996.
- Image Access Services Specification, Version 0.9 Platinum - Revisions based upon comment, April 24, 1996.
- Image Access Services Specification Version 1.0 - ICCB Configuration-controlled, pilot operational specification for contractor and commercial prototyping and interoperability testing, June 20, 1996.
- Image Access Services Specification Version 1.1 -
  Revised to remove TBR's and TBD concerning the PNF and IDF. Released for comments 6 Dec, 1996
- Image Access Services Specification Version 1.1 -
  Submitted to CCB Dec 20, 1996.

Planned Releases

- Image Access Services Specification Version 2.0 - Development engineering specification for guidance review by standards working groups and certification testing, November (TBR) 1996.

Preface

This document defines common interfaces for the United States Imagery System (USIS) image access services.

In the short term, this is a pre-operational specification activity, that is defining the client interfaces for the image access services. This will support interoperability testing of clients and implementations.

The short term goal is to test alternative implementations of clients and library implementations supporting common software interfaces, so that the community is able to demonstrate interoperability (for image access) among multiple independently developed USIS software components. The lessons learned from the pilot implementations will be incorporated into the specification prior to submission for review and feedback through the standards process (CIIWG and ISMC).

The early releases (prior to 2.0) are intended for review and prototype implementation. This specification will change over the course of this activity and follow-on activities that lead toward operational capabilities. Extensions to the specification are anticipated to address new capabilities and evolving user needs.

This specification was prepared consistent with industry practices and is modeled after those being prepared by the Object Management Group (OMG) industry consortium. This approach is consistent with guidelines and direction established by the CIO Common Imagery Interoperability Working Group (CIIWG).

Table Of Contents

1. *Overview*

1.1. Background

The Image Access Services (IAS) specification addresses the core interoperability requirements of the United States Imagery System (USIS) for client access to imagery and imagery-based products (referred to collectively as *image products*). The supported operations include image product discovery, metadata attribute retrieval, whole product retrieval, image region retrieval, and client product creation.

This IAS specification defines the interface requirements for the following facilities from the Common Imagery Interoperability Facilities (CIIF) reference model:

1. Image Access Facility (IAF)
2. Catalog Access Facility (CAF)
3. Profile & Notification Facility (P&NF)

The Image Access Facility (IAF) defines interfaces for retrieval of image products. The range of supported products includes full frame images, image chips, subimages, and display regions. The facility also supports the creation (uploading) of new products by the client.

The Catalog Access Facility (CAF) defines interfaces for query-based discovery of image products and retrieval of metadata attributes. Supported queries include attribute-based boolean queries and geographic queries.

The Profile and Notification Facility (P&NF) enables clients to create and manage interest profiles that serve as standing catalog search specifications. The standing requests allow users to register their notification preferences, so that the facility implementation can detect when new catalog entries satisfy their profile.

1.2. Facilities Overview

Figure 1-1 establishes architectural relationships within the IAS specification. A key architectural constraint is that the specialized facilities rely on the more general facilities, but not vice versa. The specialized facilities will contain details that are unique to imagery users; whereas, the general facilities will be more flexible and reusable.

Figure 1-1 Image Access and Catalog Access comprise four facilities, two general purpose and two imagery-specific

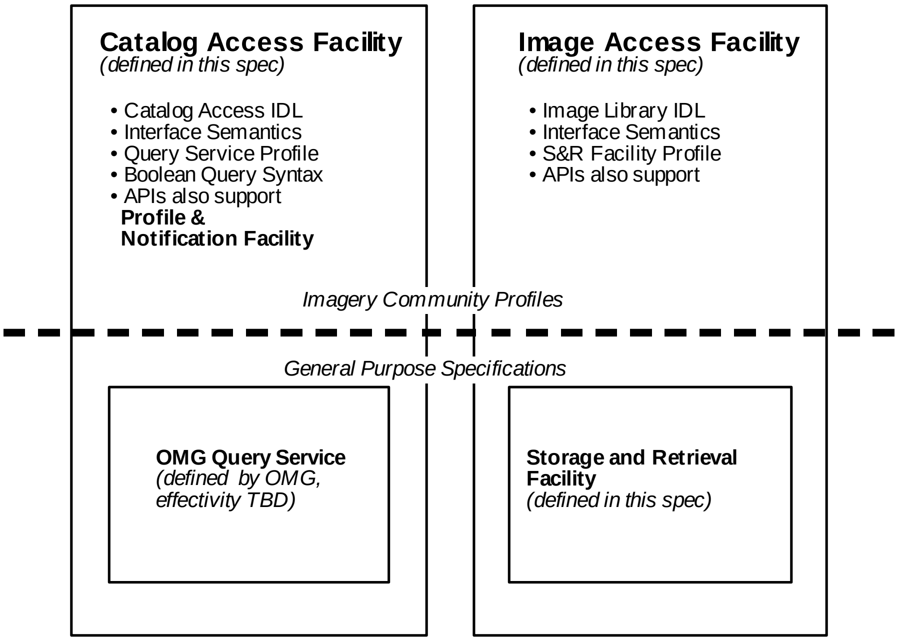

Image access services interfaces have been partitioned into a generic Storage and Retrieval Facility and an imagery-specific Image Access Facility (Figure 1-1). Image Access and Catalog Access contain specialized definitions and interfaces that directly address the needs of the imagery community interoperability.

There is one additional CIIF facility defined in this document. The Profile and Notification (P&NF) Facility reuses the CAF interfaces and defines its own interfaces as extensions. For the reused operations, the semantics are different. In the CAF, interfaces are used for one-time queries. In the P&NF, interfaces are used to post standing queries.

The IAS facilities are specified using the OMG Interface Definition Language (IDL) [5]. IDL is a language-independent notation for specifying software interfaces. IDL can be readily compiled into software interfaces for various programming languages including C, C++, Ada95, and Smalltalk.

1.2.1. Storage and Retrieval

The S&R facility principal interfaces are shown in Table 1-1. The interfaces solve generic problems, that could apply to any domain of information retrieval. Image products comprise a wide range of data types, i.e. images, text, graphics, audio, video, and multimedia.

Table 1-1 Storage & Retrieval Facility Interfaces

Interface	Purpose	Primary Clients
ProductRequest	Storage and retrieval of whole information products (full images, partial images, i.e. chips, and various exploitation products -- in file formats)	All storage and retrieval clients, browsers, viewers, editors, exploitation systems, i.e. ELTs, etc.
ArrayRequest	For array-structured information products, retrieval of the specific element values, for example image tiles.	Information product viewers with optimized image transmission and storage needs, exploitation systems, i.e. ELTs, etc.

1.2.2. Image Access

The Image Access Facility (IAF) incorporates the S&R facility, and adds the interface details necessary for interoperability and functionality for the imagery community.

The interfaces provided by IAF (in addition to those reused from the S&R facility), are shown in Table 1-2 below.

Table 1-2 Image Access Facility Inter faces

Interface	Purpose	Primary Clients
Server	Management of the connection between client and IAF/CAF servers	All image product clients, exploitation systems, etc.
IA (Image Access)	Imagery specific extensions to the S&R facility to ensure imagery interoperability and functionality	All image product clients, exploitation systems, etc.
ImageProduct	To provide a image product reference used for whole product retrieval	Clients and services accessing image products
ImageArray	To provide an image product reference used for tiled retrieval	Image product clients and services accessing image regions

Parameters	Managing parameters and specialized metadata directly associated with particular objects	Clients that access and control parameters and specialized metadata

1.2.3. Catalog Access

The Catalog Access Facility (CAF) defines interfaces for image product discovery and attribute metadata access. The interface for the CAF is shown in Table 1-3.

Table 1-3 Catalog Access Facility Interfaces

Interface	**Purpose**	**Primary Clients**
CA (Catalog Access)	Discovery of image products involving geographic search and boolean queries; retrieval of imagery metadata attributes	Image product clients including browsers, viewers, exploitation systems (ELTs), and forms-based user-interface applications

This specification includes a boolean Query Syntax for use with the CAF which is defined in Section 6. The boolean Query Syntax simplifies and decouples client software from changing community metadata and library-specific attributes.   It also enables direct utilization of community specified attributes for the purposes of querying image catalogs. [18]

1.2.4. Profile & Notification

The Profile and Notification Facility (P&NF) is a specialization of the Catalog Access Facility for the purposes of posting standing queries. The interface for the P&NF is identified in Table 1-4.

Table 1-4 Profile and Notification Facility Interfaces

Interface	**Purpose**	**Primary Clients**
PN (Profile & Notification)	Posting standing queries for future satisfaction.	Image product clients including browsers, viewers, and forms-based user-interface applications

1.3. *Interface Hierarchy*

The interface hierarchy shows how the interfaces defined in the facilities reuse each-other's definitions through inheritance. Figure 1-2 shows the interface hierarchy. Each box represents an interface type from Tables 1-1 through 1-5. The interface types are identified by their type name in bold text. Below the interface types are the operations defined in the Interface Definition Language (IDL) definitions. Inheritance relationships are shown by arrows, with the end of the arrow pointing to the inheriting interface class.

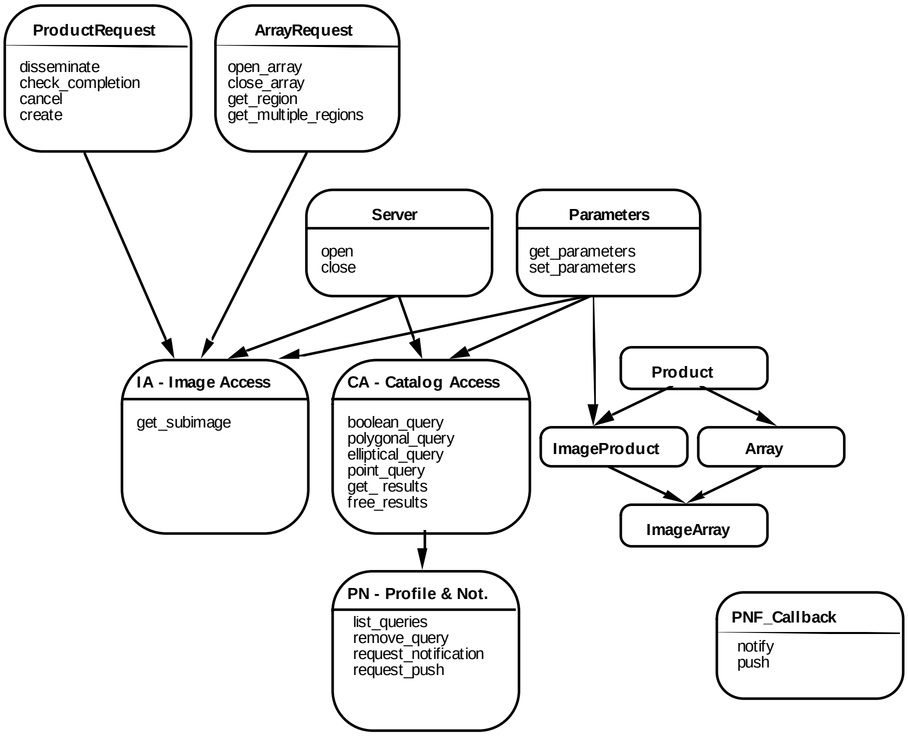

Figure 1-2 The Image Access Services Interface Hierarchy
[This hierarchy illustrates the commonality and specializations in interface architecture—arrows indicate specification inheritance]

There are two principal interfaces that clients use to access images: IA (Image Access) and CA (Catalog Access). These interfaces are defined in the Image Access Facility and the Catalog Access Facility sections 4 and 5, respectively. The Image Access interface provides operations for transfer of image products. The Catalog Access interface provides operations for image product discovery and attribute retrieval. Using these interfaces, clients obtain product references from the catalog and use them to retrieve images from the library.

The server and parameters interfaces are abstract interfaces inherited by both IA and CA interfaces.   PN (Profile & Notification) is a specialization of the CA interface.

The product interfaces (Product, Array, ImageProduct, and ImageArray) provide a capability for referencing information products. The Product interface is the most general form of reference. Specialized references exist for Array products, ImageProducts, and ImageArray products.

The above interfaces are discussed further in Section 2. Their specifications are contained in Sections 3 through 7.

2. Interface Overview

2.1. Storage & Retrieval Facility

The Storage & Retrieval Facility provides general purpose capabilities for access to stored information, such as image products.

2.1.1. Product Request Interface

The product request interface addresses the transfer of whole information products. Whole products comprise information products stored in file formats. For example, these may be full images (with or without associated headers), image chips, or other image products.

To retrieve a product, a request is made for product transfer, then the processing of the request occurs in the background through a mechanism such as the File Transfer Protocol (FTP). Other comparable mechanisms include HTTP and the OMG Data Interchange Facility. The choice of transfer mechanism is a property of the implementation, not of the interface specification. Standards requirements for the product transfer mechanisms are defined by the USIS Standards and Guidelines [1]. There is a quality of service associated with each request, which allows the client to specify delivery requirements which can be either immediate or queued.

The retrieval interface provides a basic capability for confirmation of receipt through the request identifiers and the completion status checking operation.

This interface also provides basic capabilities for creating new products (i.e. uploading to the library) from appropriately authorized clients. The product request interface is not intended to be a complete data management interface.

2.1.2. Array Request Interface

The array request interface is a general-purpose interface for the efficient retrieval of image regions and other array data. The array request can be used with an arbitrarily large source image.

Only a subset of the items in the library contain array information which can be retrieved using the array request interface. Those which do, are identified as such in the catalog metadata. The catalog will identify which products support array access. For example, very large images can be efficiently accessed using array access.

Since array retrieval does not transfer whole products, this interface necessitates that the catalog metadata contain descriptive attribute information equivalent to the image and product file header information, including references to related products.

2.2. Image Access Facility

The Image Access Facility (IAF) provides capabilities for clients to access and retrieve image products from image libraries. IAF reuses and extends the S&R interfaces. Image access adds imagery specific definitions that populate and extend the S&R facility.

Image access and catalog access are related and synergistic. Catalog access usually precedes image access, returning product references for image product retrieval. Catalog access enables image product discovery and provides access to metadata describing image products. Once the image is identified and its attributes are known, image access provides a means to retrieve the data. The catalog serves as an enabling facility for identification of image products of interest.

The purpose of image access is to provide a means to retrieve image products, both in whole and in part.

Image access provides two primary modes of retrieval. The first mode is retrieval to a specific location, which may be a pathname or other form of address. This form of retrieval is limited to whole product transfers.

The second mode enables transfer of partial image products, i.e. array regions, in response to client requests.

2.2.1. Server Interface

The Server interface is an abstract interface; it is not intended for standalone implementation. This interface contains specifications for basic connection management to the image library and catalog. There are provisions for future security extensions, which are described in Appendix E.

2.2.2. Parameters Interface

The Parameters interface is an abstract interface inherited by other interfaces. This interface has the capability to retrieve and change parameter values that are closely related to particular objects and particular client interactions with the objects. For example, the parameters can include metadata that describes the object. Parameters can also include some controllable client-specific characteristics that are used to modify the behavior of other operations, such as retrieval of array regions.

2.2.3. Image Access Interface

The Image Access interface reuses definitions from the Server interface, Parameters interface and the S&R facility. This interface adds specializations of the S&R that are appropriate for imagery applications. These specializations use the flexibility of S&R to assure a level of interoperability not available in a more generic specification.

The Image Access interface contains extensions to S&R that are imagery specific. These IAF interfaces and conventions apply across all imagery systems, and form a common basis for image retrieval and client update to image libraries.

A key imagery-specific capability in IAF is subimage retrieval. An operation is provided for retrieving a subimage based upon a geographic region in an existing image array product. The subimage will image regions that provide coverage of the requested geographic area.

2.3. Catalog Access

Catalog access provides a means of discovering imagery products and retrieving their metadata attributes. The Catalog Access Facility (CAF) specifies the APIs and data passing conventions for client access to image catalogs. The key components of CAF are discussed below.

2.3.1. Catalog Access Interface

The Catalog Access interface contains imagery-specific interfaces for searching catalog metadata. Two forms of query constraints are provided: attribute-based and geographic. Attribute-based queries are expressed using the boolean Query Syntax (See Section 6). The Catalog Access APIs include capabilities for geographic querying in several forms (polygon, ellipse, and point), that are suitable to satisfy the range of geographic search needs in the imagery community.

2.3.2. Boolean Query Syntax

The boolean query syntax is a means of specifying attribute-based search expressions. The boolean query syntax is used for image catalog queries.

The boolean Query Syntax is included in this specification to:

- Provide uniform query syntax and views
- Simplify client and catalog processing

- Decouple clients from physical schema and catalog implementations
- Associate queries directly with attribute sets, and
- Assure interoperability

2.4. *Profile & Notification Facility*

The P&NF facility enables clients to register standing queries on the image catalog. These standing queries comprise a user profile. The facility supports the notification of clients when new catalog entries match the profile's queries.

The P&NF is a specialization of the Catalog Access Facility.

2.5. *Additional Aspects*

2.5.1. *Product and Array References*

An object reference called "Product" is defined in the IDL. Its purpose is to provide robust references to arbitrary information products in a distributed environment. The role of a product reference is equivalent to a Universal Resource Locator on the Internet. The product reference might even have a hyperlink encoded in its representation, but this is an implementation choice, not a requirement of this specification.

A Product reference is an opaque structure which contains any necessary information needed to locate and retrieve the product, such as library location information and file path names. Since the Product reference contains all the necessary information for retrieval in a distributed system, distributed processing issues can be handled transparently by infrastructure software.

The product reference can have two forms: externalized and internalized. The externalized form can be stored persistently, and reside in the catalog. The internalized form is the actual product reference, it is an opaque structure.

Externalized product references are also called "stringified" references because they are represented as strings. A general purpose function, string_to_object(), will be provided by the infrastructure to convert a stringified object reference to an internalized object reference. Another predefined function, object_to_string() is used to externalize the object references.

An Array reference corresponds to an image product which contains actual imagery data, suitable for retrieval as regions (i.e., tiles). The array reference must refer to a unique image. If several images are contained within an image product, there must be a separate array reference to each enclosed image, although the complete product would have a single ordinary "Product" reference.

Product references and array references are specialized in the IAS interfaces for use by image libraries. The specialization includes support for Parameters (Section 4.2.3).

2.5.2. Exception Information

A set of arguments called `ExceptionInfo` is returned from each operation which terminates abnormally with a user-defined exception. User-defined exceptions are error conditions which are explicitly defined in the image access services IDL.

There is also a standardized set of general purpose exceptions defined by industry which address the most common reasons for failures, including communication and network errors [5] (Appendix D).

If an operation completes successfully, the exception value includes a completion status of “COMPLETED_YES” [5]. This is a positive confirmation that the requested service achieved satisfactory completion.

3. Storage & Retrieval Facility

This section defines the IDL, semantics, and sequencing of the Storage & Retrieval interfaces in detail. Each definition includes a corresponding IDL segment which is in the following typeface:

```
// Example IDL segment - An IDL comment
```

3.1. Storage and Retrieval Module

```
module InfoSandR {
```

Module “InfoSandR ” provides an enclosing scope for all of the Storage & Retrieval facility (S&R) interfaces, type definitions, exception definitions, and operation signatures.

3.1.1. Exception Information

```
#define ExceptionInfo any exception_info;
```

The ExceptionInfo symbol is used in all the exception definitions in the facility to give these exceptions a consistent set of return values. In S&R, this is a generic type ‘any’ that is intended to be specialized for specific applications. The IAF and CAF specializations are described in Section 4.2.1 and 5.1.2, respectively.

In these facilities, user exceptions are defined for all cases of bad input parameters, as well as error conditions which are unique to particular operations.

There is also a set of standard exceptions defined which cover most generic error conditions, such as communication failure (COMM_FAILURE) and lack of permission (NO_PERMISSION). The standard exceptions are listed in Appendix D.

3.2. Product Request Interfaces and Types

3.2.1. Type Definitions

3.2.1.1. Product References

```
interface Product {};
```

This is an opaque reference to any kind of information product as described in Section 2.5.1. It has no predefined operations on its interface.

The product reference is intended to be specialized and extended. The specialization for IAF/CAF is described in Section 4.2.4

3.2.1.2. Storage Specifications

```
typedef any Loca tionSpec;
typedef sequence<LocationSpec> LocationSpecList;
```

A location specification contains information necessary for the transfer of stored information. The contents include the necessary information for accessing information transfer mechanisms such as FTP, HTTP, or the OMG Data Interchange Facility. The actual contents are defined by specializations of the S&R facility.

The LocationSpecList sequence is a variable length list of location specifications. A location specification can be used to denote the source for information or the destination for information.

3.2.1.3. Exceptions

```
exception BadProductReference { ExceptionInfo };
exception BadLocationSpec { ExceptionInfo };
```

These exceptions are returned by operations that use product references and locations as input parameters. When returned, these exceptions indicate that the corresponding arguments were invalid. Additional explanations may be contained in the ExceptionInfo.

3.2.1.4. Request Identifiers

```
typedef string RequestId;
typedef sequence<RequestId>   RequestIdList;
```

The request identifier is a mechanism for tracking requests. The server implementation generates the request identifier, which is unique for each request. The RequestIdList type is used to convey a set of request identifiers.

3.2.1.5. Name Values

```
    struct NameValue { string name; any value; };
    typedef sequence<NameValue> NameValueList;
```

```
typedef sequence<string> NameList;
exception BadName { ExceptionInfo };
exception BadValue { ExceptionInfo };
```

The NameValueList is a generally useful structure for conveying named attribute values. The NameList sequence is a sequence of identifiers used as an argument to convey a list of names corresponding to a NameValueList data type.

The BadName exception is returned when a name argument is invalid. The BadValue exception is returned when an invalid value argument is detected.

3.2.1.6. Response Service

```
enum ResponseService { IMMEDIATE, QUEUED };
exception ResponseServiceNotAvailable { ExceptionInfo };
```

Qualities of service are defined by the ResponseService enumeration. The values are IMMEDIATE and QUEUED. The IMMEDIATE service is defined as "best effort" performance. The queued service defines a lower priority "when possible" effort , the status of which can be assessed by the check_completion operation.

If a particular quality of service is not available from the server implementation, it may raise the ResponseServiceNotAvailable exception. If a server only implements a single quality of service, it is defined to be the IMMEDIATE mode.

```
exception  TooManyRequests { ExceptionInfo };
```

Some server implementations may have limitations on the number of outstanding requests. This exception is an error return indicating that this implementation limit has been exceeded.

```
exception ProductUnavailable { ExceptionInfo };
```

This exception indicates that the server is temporarily unable to provide array operations on the product requested. This condictions might occur if for instance the product was in offline storage.

3.2.2. Product Request Interface

```
interfac e ProductRequest {
```

The ProductRequest interface defines a particular type of information storage and retrieval server which handles whole product transfer requests.

3.2.2.1. Disseminate Operation

```
RequestIdList disseminate(
    in Product product_to_disseminate,
    in LocationSpecList destinations,
    in ResponseService service )
    raises ( BadProductReference, BadLocationSpec,
        ResponseServiceNotAvailable, TooManyRequests )
    context(  “ContextInfo” );
```

The disseminate operation requests the initiation of the transfer of a whole product. The Product argument is a reference to the particular product to transfer. The LocationSpecList argument is a list of destinations for the product. This list may include destinations for the requesting client as well as third-party clients; thus this operation supports a form of push-mode transfer through a third-party request. The ResponseService is a quality of service specification as described above. The return value, a RequestIdList, provides a unique identifier (from the server) for each of the listed destinations. RequestIds are used for tracking the progress of the request using the completion checking operation. The ProductRequest implementation returns the thread of control to the client as soon as possible after this request is invoked in order to process this operation in the background.

The disseminate operation will return the BadProductReference exception if the product is not present or the reference is invalid. The BadLocationSpec exception is returned when one or more locations are invalid. (Note: Prescreening of LocationSpecs is not required by the Image Access Facility specialization of this operation).

The ResponseServiceNotAvailable exception is returned if the server does not support the requested quality of service. The TooManyRequests exception is returned if the server’s implementation limit is exceeded for the number of outstanding requests.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

3.2.2.2. Completion Checking Operation

```
enum CompletionState
    { COMPLETED, IN_PROGRESS, ABORTED, CANCELED,
      PENDING, OTHER };
```

```
exception BadRequestId { ExceptionInfo };
CompletionState check_completion(
    in RequestId request_identifier,
    out string state_information )
    raises ( BadRequestId )
    context(  “ContextInfo” );
```

The completion checking operation enables clients to check the status of a request. The request is uniquely identified by the request identifier argument. A CompletionState enumeration argument is returned indicating the completion status. Any additional explanation is contained in the state_information argument.

The completion states are described as follows. The state COMPLETED is returned if a normal successful termination has already occurred. The state IN_PROGRESS is returned if the request is in an active state of execution and has not yet encountered any error conditions. The state ABORTED is returned if an error condition has caused an abnormal termination. The state CANCELED is returned when the request has been canceled by a previous request. The state PENDING is returned if a request is known and pending for future service, but the transfer has not started. The PENDING state only applies to requests submitted with a ResponseService mode of QUEUED. The state OTHER indicates that the request is in some state of completion, not indicated by the above states. In all cases, the string state_information may return additional information.

The BadRequestId exception is returned if there is an invalid request identifier.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

3.2.2.3. Cancel Operation

```
CompletionState cancel(
    in RequestId request_identifier )
    raises ( BadRequestId )
    context(  “ContextInfo” );
```

The cancel operation enables clients to terminate an outstanding request initiated by other operations, such as disseminate and create. The request identifier argument uniquely identifies the request concerned. A CompletionState is returned with complete information about the status of the request.

The operation may return a BadRequestId exception if the RequestId argument is invalid.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

3.2.2.4. Create Operation

```
exceptio n BadCreationAttributes { ExceptionInfo };
Product create( in LocationSpec initial_product_data,
    in NameValueList creation_attributes,
    out RequestId request_id)
    raises ( BadCreationAttributes, BadLocationSpec )
    context(  “ContextInfo” );
```

The create operation stores a new product in the library. The operation generates a new product reference as its return value. The source data for the new product is indicated by the LocationSpec argument. The request identifier allows the tracking of the request through the completion checking operation.

The creation attributes argument contains additional information necessary for creation. For example, this argument can contain additional metadata if required.

A RequestId is returned as an output parameter. The RequestId can be used to check completion or cancel the request. The new product reference should not be used until the request has completed. Otherwise an exception, such as the standard exception OBJECT_NOT_EXIST, may be returned.

The BadCreationAttributes exception is returned if the creation attributes (such as metadata) are invalid. The BadLocationSpec exception is returned if the LocationSpec argument is invalid.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

3.3. Array Interfaces and Types

3.3.1. Array Product Reference

```
interface Array: Product  {};
```

The Array product reference defines a specialized type of product reference, that is reserved for products which contain array data (such as image pixels) and can support array data retrieval.

3.3.2. Element Type Enumeration

```
enum ElementType { BITDATA, BYTEDATA, SBYTEDATA, INT2DATA,
    SINT2DATA, INT4DATA,  SINT4DATA,  FLOAT4DATA, COMPLEXDATA,
    FLOAT8DATA, OTHERDATA };
```

Array retrieval comprises direct access to array elements (or pixels); the various element types are defined here. They include binary (BITDATA), unsigned bytes (BYTEDATA), signed bytes (SBYTEDATA), unsigned short integers (INT2DATA), signed short integers (SINT2DATA), unsigned long integers (INT4DATA), signed long integers (SINT4DATA), floating point (FLOAT4DATA), complex numbers (COMPLEXDATA), double precision floating point (FLOAT8DATA), and other miscellaneous representations (OTHERDATA).

3.3.3. Buffer Union Type

```
union Buffer switch (ElementType) {
    case BITDATA: sequence<octet> bit_data;
    case BYTEDATA: sequence<octet> ubyte_data;
    case SBYTEDATA: sequence<char> byte_data;
    case INT2DATA: sequence<unsigned   short> ushort_data;
    case SINT2DATA: sequence<short> short_data;
    case INT4DATA: sequence<unsigned long> ulong_data;
    case SINT4DATA: sequence<long> long_data;
    case FLOAT4DATA: sequence<float> float_data;
    case COMPLEXDATA: sequence<float> complex_data;
    case FLOAT8DATA: sequence<double> double_data;
    default: sequence<octet> other_data; };
```

Regions of each element type have a representation as an IDL sequence in this union. All array regions can be represented by values of this union type.

The listed data types are a superset of the elementary types for pixels defined in ISO IPI [19].

3.3.4. Region Specification Structure

```
typedef any RegionSpec;
typedef sequence<RegionSpec> RegionSpecList;
```

The RegionSpec identifies a particular region within an array (or image). The choice of parameters is application defined. The contents of RegionSpec attributes are used for only one request. Other attributes established for use across multiple requests are established by another approach.

3.3.5. Region Data Structure

```
struct RegionData {
    RegionSpec region_spec;
    NameValueList region_header;
    ElementType element_type;
    Buffer region_data;
    };
```

The region data structure contains the elements (or pixels) of the region and some self-descriptive information. The RegionSpec member describes the region and may differ from the input parameters specified in a request. The ElementType member defines the representation of the elements (or pixels). The Buffer member contains the element values. By default, element values will be stored in row-major order in the Buffer sequence.

Attributes returned in the region_header describe the characteristics and format of the returned data. For example, a region thickness may be identified for the number of ElementType values correspond to each pixel.

Thick pixels (such as color images) may be interleaved in the sequence or not based upon another attribute for interleaving (For example, controlled through the IAF Parameters Interface).

3.3.6. Array Request Interface

```
interface  ArrayRequest {
```

The Array Request interface includes capabilities for retrieval of region data.

3.3.6.1. Open Array

```
                        exception ProductUnavailable
{ExceptionInfo};
        void open_array(in Array product, in any access_kind)
            raises ( BadProductReference, ProductUnavailable )
            context(  “ContextInfo” );
```

The operation initiates access to an array object. The product argument identifies the product to be opened. A call to open_array is required for an array product before other operations using the Array reference are invoked. Note that is method is in addition to the open method on the server. To access an array, the client must first open the server in which the array resides and then open the array itself.

The AccessKind argument is application defined, and may, for example indicate if the array is writable. A reserved value of this argument contains a nil pointer for the value field of the type any. This reserved value can be used safely by clients without conflicting with application defined values.

A BadProductReference exception will be returned if the array product reference is. The exception ProductUnavailable is returned if the server is temporarily unable to open the array due to internal server state (i.e. product is in offline storage). The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

3.3.6.2. Close Array

```
exception ArrayNotOpen { ExceptionInfo };
void close_array(in  Array product)
    raises ( BadProductReference,ArrayNotOpen)
    context(  “ContextInfo” );
```

The close operation indicates that the client has completed access to an array product. The operation close_array is the converse of the operation open_array. The operation close_array is executed after open_array and any other ArrayRequest operations using the array reference.

The array product reference argument indicates the product to be closed. The operation may return the BadProductReference exception if the array product reference argument is invalid. The ArrayNotOpen exception is returned if the array product has not been previously opened by this client. The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

3.3.6.3. Get Region Operation

```
exception BadRegionData { ExceptionInfo };
exception BadRegionSpec      { ExceptionInfo };
void get_region(
    in Array product,
    in RegionSpec region_spec,
    inout RegionData region_data )
    raises( BadProductReference, BadRegionSpec,
        BadRegionData, NotOpen )
    context(  “ContextInfo” );
```

This operation can be used to retrieve element (or pixel) data from an array product. The Array argument is the product of interest. The RegionSpec argument identifies the region within the product. The get region operation has an inout argument, which is a region data structure. It is an inout argument, indicating that the client allocates storage for the result. The returned RegionData structure contains the image pixels and self-descriptive information.

The BadProductReference exception is returned if the product reference is invalid. The BadRegionSpec exception is returned if the region specification is invalid.

An exception, BadRegionData is returned if the data is unavailable in the appropriate compression or data format. The ArrayNotOpen exception is returned if the array product has not been previously opened by this client.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

3.3.6.4. Get Multiple Regions Operation

```
typedef sequence<RegionData> RegionDataList;
void get_multiple_regions(
    in Array product,
    in RegionSpecList region_specs,
    inout RegionDataList region_data_list )
    raises( BadProductReference, BadRegionSpec,
        BadRegionData, ArrayNotOpen )
    context( “ContextInfo” );
```

This operation retrieves one or more array regions to memory areas indicated by the RegionDataList. For example, this operation may be used to retrieve multiple image tiles, each specified as a separate region.

The array product reference argument identifies the source array for region retrieval. The RegionSpecList argument contains the requested specifications for each of the retrieved regions. The RegionDataList defines the allocated member array, where the region data will be returned.

The BadProductReference exception is returned if the array product reference argument is invalid. The BadRegionSpec exception is returned if one or more RegionSpecs in the RegionSpecList argument are invalid. The BadRegionData exception is returned if the data is unavailable in the appropriate compression or data format. The ArrayNotOpen exception is returned if the array product has not been previously opened by this client.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

4. Image Access Facility

This section defines the semantics and sequencing of the Image Access Facility interfaces in detail.

4.1. S&R Profile

The Image Access Facility (IAF) incorporates the definitions of Storage and Retrieval (S&R) to provide interoperability between clients and image libraries. This section defines conventions for use of the S&R facility by imagery systems and their clients.

In this section, the semantics of the disseminate and create operations imply an immediate return without requiring checking of the validity of the location specification arguments.

4.1.1. The Create Operation

To create a new product, the client must be able to specify the new product in a file format which includes all appropriate metadata or specify metadata separately in the create request (using the NameValueList argument). This assures that all needed metadata is provided to the library server to allow creation of the new product and incorporation of applicable metadata in the library catalog. This will allow subsequent image product discovery and retrieval.

When the create operation is called, it issues a unique product reference that is only initially known to the requesting client. The client should verify completion of the creation request using the completion checking operation before utilizing the product reference.

4.2. Image Access Services Module

```
module IAS {
```

Module “IAS ” provides an enclosing scope for all of the Image Access Services interfaces, type definitions, exception definitions, and operation signatures.

By convention, all data types and exceptions are defined at the module level. Depending on the language binding used, the characters “IAS ” will appear in the scoping prefix to the these identifiers. All operation signatures are declared in particular interfaces in the facilities. These scoping prefixes will have both module and interface identifiers. For example, IAS::IA for image access facility operations.

4.2.1. Data Types

```
struct ImageExceptionInfo {
    short status _code;
    string status_text;
    string exception_type;
    };
```

The ImageExceptionInfo structure is supplied by all IAS implementations to populate the exception_info value returned by all operations. Use of the member fields is implementation defined. Implementations of IAF will use this structure for returning user exceptions.

```
typedef string ClientContext;
```

The ClientContext is implementation-defined information.

4.2.2. Server Interface

```
    interface Server {
```

The Server interface is an abstract interface that contains some common operations which are supported by all IAS servers, such as establishing and terminating client connections.

4.2.2.1. Exceptions

```
        exception AlreadyConnected { ExceptionInfo };
```

This exception is returned when an attempt is made to reconnect to an already connected server.

```
        exception BadOpenCriteria  { ExceptionInfo };
```

This exception indicates that the connection is denied because the specified criteria is unacceptable.

```
        exception NoConnectionEstablished { ExceptionInfo };
```

The NoConnectionEstablished exception is returned when there has been no previous successful open operation on this server by this client.

4.2.2.2. Open Operation

```
ClientContext open(in NameValueList open_criteria)
    raises ( AlreadyConnected  , BadOpenCriteria)
    context(  “ContextInfo” );
```

The open operation must be invoked before any other operation are invoked on that server in order to establish a connection between the client and server. The connection establishes some client-specific state information in the server, such as the resetable parameters.

The argument includes a flexible list of arguments represented as a NameValueList type. This list contains any information required by the server to identify and authorize access. These values must be known and acceptable to the server before a connection is granted. Successful return indicates that the connection was granted; otherwise an exception is returned.

The operation has a return result of data type ClientContext. The contents and use of this value is application specific.

The AlreadyConnected exception is returned by the server when there is already an established connection by the requesting client. The BadOpenCriteria exception is returned when the open criteria is not accepted by the server. The standard exception NO_PERMISSION is returned if the client does not have sufficient permissions to access the server.

4.2.2.3. Close Operation

```
void close()
    raises ( NoConnectionEstablished )
    context(  “ContextInfo” );
```

The close operation is invoked after the completion of other invocations to a server interface. The client must reconnect (using the open operation) before it can successfully invoke other operations.

The NoConnectionEstablished exception is returned when there has been no previous successful open operation on this server by this client.

4.2.3.   Parameters Interface

```
interface Parameters {
```

The Parameters interface contains definitions which provide access to parameter information directly associated with an object or a particular client’s interaction with an object. Client-specific *resettable* parameters establish state information in the server which affects subsequent requests.

4.2.3.1. Get Parameters Operation

```
    void get_parameters(
            in InfoSandR::NameList
names_of_parameters_requested,
            out InfoSandR::NameValueList parameter_values)
        raises( InfoSandR::BadName )
        context(  “ContextInfo” );
```

The get parameters operation allows the retrieval of attributes which are directly stored or closely associated with an object, for example an image array or a particular IAS server object.

The NameList argument identifies the parameter names to be retrieved as output arguments to this invocation. Each name in this list identifies one parameter. The NameValueList output parameter will contain the values of the parameters. Each element in this list contains the parameter name and its corresponding value. The parameters requested may be returned in the NameValueList in an arbitrary order.

The BadName exception is returned if one or more names in the NameList argument are invalid. Further details on which name(s) were invalid should be returned in ExceptionInfo.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

4.2.3.2. Set Parameters Operation

```
    exception CannotSet { ExceptionInfo };
    void set_parameters(
            in InfoSandR::NameValueList parameter_values)
        raises( InfoSandR::BadName, InfoSandR::BadValue,
            CannotSet )
        context(  “ContextInfo” );
```

The set_parameters operation modifies the value of client-specific resettable parameters which are used to affect subsequent invocations of other operations.

The object implementation can substitute default values if the parameters are not initialized by the client.

The NameValueList parameter is a list of parameter names and their corresponding values.

The BadName exception is returned if one or more names in the NameValueList argument are invalid (for example, they do not correspond to known parameters). . Further details on which name(s) were invalid should be returned in ExceptionInfo. The BadValue exception is returned if one or more values in the NameValueList argument are not acceptable by the object implementation (either because they are the wrong type or exceed acceptable value ranges). The CannotSet exception is returned if one or more parameters identified in the NameValueList argument are not resettable.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

4.2.3.3. Common Parameters

There is only one parameter which is explicitly defined in this IAS specification. All other parameters are defined within the appropriate profile.

There are two types of parameters: read-only and resettable. The read-only parameters are not client-specific. The resettable parameters are managed on a per-client basis by the server implementations. These parameters embody client-specific state information which affect the processing of operations.

The only parameter defined by the IAS specification is the parameter “ParameterNames. All implementations of the Parameters interface will have a parameter called "ParameterNames" which is metadata that enables discovery of the available parameters supported by that implementation. The representation is a NameValueList data type. The name field of each NameValue struct represent the parameter name. The value field of each NameValue struct is a boolean that indicates whether the parameter is resettable i.e. TRUE means parameter is resettable.

4.2.4. Image Product and Image Array

```
interface ImageProduct: InfoSandR::Product, Parameters {};
interface ImageArray: InfoSandR::Array, ImageProduct {};
```

IDL interfaces are used to declare the object reference types for image products and array products. These references are specializations of the generic references defined in the S&R facility for Product and Array. Both of these types add the operations for parameter management, so that some attributes specifically associated with these objects can be retrieved without going to the catalog.

An ImageProduct reference corresponds to a whole product which may be retrieved and created using the InfoSandR::ProductRequest interface with Image Access specializations.

Regions from an ImageArray may be retrieved using the InfoSandR::ArrayRequest interface with Image Access specializations. Tiles from an ImageArray may be retrieved using the ArrayRequest::get_multiple_regions operation. Subimages from an ImageArray may be retrieved using the IA::get_subimage operation.

4.2.5. Image Access Interface

```
interface IA: Server, Parameters,
    InfoSandR::ProductRequest, InfoSandR::ArrayRequest {
```

The Image Access interface contains definitions which provide imagery-specific extensions to the S&R facility.

4.2.5.1. Exception Types

```
const string CompressionNotAvailable
    = "Compression Not Available";
const string FormatNotAvailable
    = "Format Not Available";
```

These exceptions types are supplied in the exception information when the appropriate compression or data format preferences indicated by the parameters cannot be supported. In particular, these types are returned with the ArrayRequest::BadRegionData exception as the exception_type member.

4.2.5.2. Image Location Specification

```
enum ImageLocationKind {PathKind, HyperlinkKind,
    AddressKind};
struct PathInfo {
    string user_name;
    string pass_word;
    string host_name;
    string path_name;
    string file_name;
    };

union ImageLocationSpec switch (ImageLocationKind) {
    case PathKind:
        PathInfo path;
    case HyperlinkKind:
        string hyperlink;
    case AddressKind:
        any address;
    };
```

This ImageLocationKind structure is supplied by IAS clients as the values of the InfoSandR::LocationSpec. The LocationSpec is used to indicate the source or destination of a whole image product transfer.

There are multiple variants. A PathKind variant designates a complete pathname within account information. This variant is suitable for use for non-anonymous FTP transfers.

The HyperlinkKind variant is a resource indicator, as commonly used on the Internet for anonymous transfers. The AddressKind is an additional variant for extensions or implementation-specific uses.

4.2.5.3. Region Parameters

```
    struct RegionParameters {
        unsigned long horizontal_size;
        unsigned long vertical_size;
        unsigned long resolution_level;
        };
```

The RegionParameters data type is used with the set_parameters and get_parameters operation. The RegionParameters affects the processing of the get_region operation of InfoSandR::ArrayRequest interface. They are defined here since they are imagery-specific.

The region parameters are used to establish some conventions for use of the ArrayRequest interface, that apply to subsequent operations until modified by another call to set_parameters. The use of the horizontal_size, vertical_size, and resolution_level indicate the size of the region in pixels and the resolution level. The affect of the set_parameters operation is that it establishes some state information in the ArrayRequest server implementation that is used for subsequent requests to the get_region operations. The set_parameters operation is invoked on a particular Image Access server, since the IA interface inherits from ArrayRequest and therefore is an ArrayRequest object.

4.2.5.4. Display and Tile Region Specifications

```
        struct DisplayRegionSpec {
            long x_region_center;
            long y_region_center;
            };
```

The DisplayRegionSpec structure is supplied as the contents of the InfoSandR::RegionSpec for the get_region operation. It is also used in the RegionData return value.

The coordinates of the region within the source image are defined by x and y region centers, which are the pixel coordinates in source image space (regardless of the current RRDS level being retrieved).

```
    struct TileRegionSpec {
        long x_region_center;
        long y_region_center;
        unsigned long horizontal_size;
        unsigned long vertical_size;
        unsigned long resolution_level;
        };
```

The TileRegionSpec structure is supplied as each RegionSpec of the InfoSandR::RegionSpecList for the get_multiple_regions operation. It is also returned in the RegionData return value from get_multiple_regions.

The coordinates of the region within the source image are defined by x and y region centers, which are the pixel coordinates in source image space. The horizontal_size and vertical_size members define the size of the retrieved region. The resolution_level member selects the RRDS level for retrieval of this region.

4.2.5.5. Get Subimage Operation

```
struct GeoCoords {
    double lat, lon; // degrees
    };
typedef sequence<GeoCoords> GeoCoordsList;
exception BadCoord { ExceptionInfo };

InfoSandR::RequestId get_subimage(
    in ImageArray image_array_product,
    in GeoCoords upper_left,
    in GeoCoords lower_right,
    in InfoSandR::LocationSpec location)
raises ( InfoSandR::BadProductReference, BadCoord,
    InfoSandR::BadLocationSpec, InfoSandR::TooManyRequests )
context( “ContextInfo” );
```

The get subimage operation causes the generation of a subimage and stores it in a specified location. The retrieved subimage will provide coverage of the bounding box specified by the geographic coordinates. The subimage has the form of a whole image product; although it is not required to be cataloged.

The server implementation returns the thread of control to the client as soon as possible after this request in order to process this operation in the background.

The subimage will be returned with the appropriate file format, compression, and RRDS levels as indicated by the appropriate parameters.

The ImageArray argument indicates the source image product from which the subimage will be created. The upper_left and lower_right arguments define a bounding-box geographic area (oriented to image boundaries). The upper_left argument is the North West corner of this area. The lower_right corner is the South East corner of this area. The LocationSpec argument defines the destination for the subimage. The server will transfer the resulting subimage to this location in response to this request. The RequestId return argument is a unique indicator for this request which can be used to monitor completion or cancel the request.

Each GeoCoords structure represents a single geographic location. The coordinates are indicated in floating point degrees of latitude (lat) and longitude (lon). The associated datum could be established as an parameter value.

A BadCoord exception indicates that one or more of the coordinates provided are unacceptable by the implementation. The implementation will attempt to provide tile coverage for the bounding box, but may return partial coverage if full coverage is not available. For example, if no coverage of the bounding box is available from this image array, then it is appropriate to return the BadCoord exception.

The BadProductReference exception is returned if the product reference is invalid. If the LocationSpec is invalid the server may return the BadLocationSpec exception. The checking of the LocationSpec argument by the implementation is not required for IAF implementations. The TooManyRequests exception is returned if the request exceeds the number of outstanding requests the server can support.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

5. Catalog Access Facility

This section defines the semantics and sequencing of the Catalog Access Facilities interfaces. Catalog access involves geographic and attribute-based search criteria.

5.1. Image Access Services Module - Continued

```
module IAS {
```

Module “IAS ” is extended here for the catalog access facility definitions. By convention, all type definitions and exceptions are declared at the module level. For clarity, some exceptions and data types are defined in the section defining the relevant operation.

5.1.1. Type Definition Query Results

```
typedef string QueryId;
typedef sequence<string> AttributeValues;
struct QueryHit {
      ImageProduct product_ref;
      AttributeValues attributes;
      InfoSandR::RegionData browse_image;
      };
typedef sequence<QueryHit> QueryHitList;
struct QueryResults {
      InfoSandR::NameList attribute_names;
      QueryHitList query_hits;
      };
```

The QueryId is a unique identifier that is used for subsequent retrieval of additional or updated query results. A QueryId remains valid only during the CA server session where the query was submitted, where a CA server session is defined as beginning with the operation “open” and ending with the operation “close”.

The QueryResults type is used for return values generated from catalog queries.

Using this definition, query results are returned in aggregate as type QueryResults. Each individual query result is type QueryHit. The query results are self-identifying since they include a NameList containing the attribute names in the order returned.

Each QueryHit includes a product reference, an optional browse image, and an attribute list. The browse image is a reduced resolution overview (or thumbnail) image.

Attribute values are returned as a sequence of strings. The result values are returned in their character-based forms. This applies for all attribute types, numeric (N), alphanumeric (A), and mixed (A/N). Each will be returned as a string of the appropriate length.

The product reference is always returned with each query hit. A browse image is optionally returned if specifically requested by setting the appropriate parameter through the parameters interface. The product request and browse image have their own member values in the QueryHit structure. The other attributes are returned in the requested order in the AttributeValues member.

5.1.2. Catalog Access Interface

```
interface CA: Server, Parameters {
```

This interface contains all of the imagery-specific operations for image catalog access, including boolean and geographic queries.

5.1.2.1. Exceptions

```
  exception BadQuerySyntax { ExceptionInfo };
  exception BadAttribute { ExceptionInfo };
  exception BadQueryValue { ExceptionInfo };
  exception BadEllipse { ExceptionInfo };
  exception BadQueryId { ExceptionInfo };
```

These exceptions are returned by the CAF operations. A BadQuerySyntax exception indicates that the query was improperly formed according to the rules defined in Section 6. A BadAttribute exception indicates that one or more of the attributes used in the query are inappropriate or unknown. A BadQueryValue exception indicates that a literal-constant value used in the query expression was inappropriate. A BadEllipse exception indicates that the axes or azimuth are inappropriate. A BadQueryId exception is returned if a QueryId is invalid.

In addition to the above semantics, the ExceptionInfo may return supplementary information which provides further indications of the problem causing the exception.

5.1.2.2. Boolean Query Operation

```
    QueryId boolean_query(
        in string boolean_query_expression )
    raises ( BadQuerySyntax, BadAttribute,
             BadQueryValue, InfoSandR::TooManyRequests)
    context(  “ContextInfo” );
```

This is an ordinary catalog search query. It takes a boolean Query Syntax (BQS) expression as input and returns a QueryId. Successful completion of this operation indicates that the submitted query was syntactically correct and was accepted for processing. The QueryId returned can be used with the get_results operation to retrieve the results.

The BadQuerySyntax exception is returned if the BQS query has illegal syntax (Section 6.3). This exception can also result if an improper operation is applied to a queryable attribute (Section 6.5). The BadAttribute exception is returned if the query expression contains an unknown or non-queryable attribute. The BadQueryValue exception is returned if a the query expression contains an invalid literal-constant value. A TooManyRequests exception is returned if the server has exceeded an implementation limit for the number of simultaneous queries.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

5.1.2.3. Polygonal Query Operation

```
exception TooFewVertices { ExceptionInfo };
exception TooManyVertices { ExceptionInfo };
QueryId polygonal_query(
        in string boolean_query_expression,
        in GeoCoordsList polygon_vertices
        )
    raises ( BadQuerySyntax, BadAttribute,
                BadQueryValue, BadCoord,
                TooFewVertices, TooManyVertices,
            InfoSandR::TooManyRequests )
    context(  “ContextInfo” );
```

A boolean query is supplemented by the specification of a polygonal shape. The CAF implementation will include products that overlap this polygonal shape and match the BQS expression. In other words, image products which overlap any portion of the polygon will match the query, as long as, the product attributes also satisfy the boolean query expression.

The first argument is the boolean query expression in BQS syntax. The second argument defines the polygon as a list of polygon vertices.

The polygon vertices must define a single closed area. The order of the vertices in the polygon_vertices argument define a counterclockwise traversal of the edges of the polygon. This establishes a convention for determining the interior of the polygon. The polygon must have no less than 3 vertices otherwise the TooFewVertices exception will be returned. The maximum number of vertices is implementation specific. If this value is exceeded, the TooManyVertices exception is returned.

Successful completion of this operation indicates that the submitted query was syntactically correct and was accepted for processing. The QueryId returned can be used with the get_results operation to retrieve the results.

The BadQuerySyntax exception is returned if the BQS query has illegal syntax (Section 6.3). This exception can also result if an improper operation is applied to a queryable attribute (Section 6.5). The BadAttribute exception is returned if the query expression contains an unknown or non-queryable attribute. The BadQueryValue exception is returned if a the query expression contains an invalid literal-constant value. The BadCoord exception is returned when one or more of the coordinates is invalid. For example, this exception will result if the vertices define a shape with multiple closed areas. A TooManyRequests exception is returned if the server has exceeded an implementation limit for the number of simultaneous queries.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

5.1.2.4. Elliptical Query Operation

```
QueryId elliptical_query(
    in string boolean_query_expression,
    in GeoCoords ellipse_c  enter,
    in double major_axis, // meters
    in double minor_axis, // meters
    in double azimuth  // decimal degrees from North
    )
raises ( BadQuerySyntax, BadAttribute,
            BadQueryValue, BadCoord, BadEllipse,
             InfoSandR::TooManyRequests)
context(  “ContextInfo” );
```

A boolean query is supplemented by the specification of an elliptical shape. The query results will include products which provide coverage of any portion of the ellipse and satisfy the BQS expression. The ellipse is defined by its center point, major and minor axes, and the azimuth clockwise rotation from North of the major axis. Circle queries use this same API by using identical major and minor axes.

The first argument is the BQS expression. The second argument defines the center point of the ellipse. The third argument is the length of the major axis of the ellipse, in meters. The fourth argument defines the minor axis of the ellipse, in meters. The fifth argument, azimuth, indicates the rotation of the ellipse.

Successful completion of this operation indicates that the submitted query was syntactically correct and was accepted for processing. The QueryId returned can be used with the get_results operation to retrieve the results.

The BadQuerySyntax exception is returned if the BQS query has illegal syntax (Section 6.3). This exception can also result if an improper operation is applied to a queryable attribute (Section 6.5). The BadAttribute exception is returned if the query expression contains an unknown or non-queryable attribute. The BadQueryValue exception is returned if the query expression contains an invalid literal-constant value.

The BadCoord expression is returned if the coordinates of the center point are invalid. For example, this exception would be returned if latitude is greater than 90 degrees or less than -90 degrees.

The BadEllipse expression is returned if the axes or azimuth arguments are invalid. A TooManyRequests exception is returned if the server has exceeded an implementation limit for the number of simultaneous queries.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

5.1.2.5. Point Query Operation

```
    QueryId point_query(
        in string boolean_query_expression,
        in GeoCoords point_geo_location
        )
    raises ( BadQuerySyntax, BadAttribute,
                BadQueryValue, BadCoord,
InfoSandR::TooManyRequests )
    context(  “ContextInfo” );
```

A boolean query is supplemented by the specification of a geographic point. Image products will be returned that match this query, i.e., those that satisfy the BQS expression and also provide coverage of this point.

The first argument is the BQS expression. The second argument defines the location of the geographic point.

Successful completion of this operation indicates that the submitted query was syntactically correct and was accepted for processing. The QueryId returned can be used with the get_results operation to retrieve the results.

The BadQuerySyntax exception is returned if the BQS query has illegal syntax (Section 6.3). This exception can also result if an improper operation is applied to a queryable attribute (Section 6.5). The BadAttribute exception is returned if the query expression contains an unknown or non-queryable attribute. The BadQueryValue exception is returned if the query expression contains an invalid literal-constant value.

The BadCoord expression is returned if the coordinates of the geographic point are invalid. For example, this exception would be returned if latitude is greater than 90 degrees or less than -90 degrees. A TooManyRequests exception is returned if the server has exceeded an implementation limit for the number of simultaneous queries.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

5.1.2.6. Get Results Operation

```
void get_results(
    in QueryId query_result_identifier,
    in long number_hits_to_return,
    out long number_of_hits_remaining,
    out QueryResults product_records )
raises ( BadQueryId)
context(  “ContextInfo” );
```

All of the CA query operations return a QueryId for every successfully submitted query. This QueryId may be used with the get_results operation to obtain the results of each query.

The operation get_results has an input argument for QueryId for the desired result set. The second argument indicates how many of the results to return. The results are returned as the QueryResults out argument and the number of hits that remain to be retrieved is returned in the third argument, number_of_hits_remaining. This operation may be called as many times as necessary to retrieve the results.

The BadQueryId exception is returned if the QueryId argument is invalid. A QueryId remains valid only during the CA server session where the query was submitted, where a CA server session is defined as beginning with the operation “open” and ending with the operation “close”.

The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

5.1.2.7. Free Results Operation

```
void free_results(
    in QueryId query_result_identifier )
raises( BadQueryId )
context(  “ContextInfo” );
```

The free_results operation notifies the catalog server that the client does not intend to retrieve any additional results for the indicated QueryId. The catalog server may free any resources allocated to the indicated QueryId, including any remaining results.

If a QueryId is returned from any of the above operations, clients will either call this operation or the get_results operation until all results are either retrieved or freed.

The input argument is a QueryId which refers to the specific result set to be freed.

A BadQueryId exception is returned if the QueryId is invalid. A QueryId remains valid only during the CA server session where the query was submitted, where a CA server session is defined as beginning with the operation “open” and ending with the operation “close”.   The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

6. Boolean Query Syntax

6.1. Overview

The boolean query syntax is a key part of the specification of the Catalog Access Facility (CAF).

To provide interoperability, this specification defines conventions on arguments as well as the operation signatures. This section defines a syntax for dynamic queries, called the boolean query syntax.

The boolean query syntax is a notation for expressing queries based directly upon pre-defined attribute lists.

The query is based on an attribute list instead of a particular schema. This simplifies the complexity of the client for query generation, and avoids constraining the design of the schema or schema view in the server implementation of the CAF.

6.2. Client Paradigm

This approach requires the server to translate the query into the appropriate schema and query language syntax, for example SQL92. Translated queries can become quite complex. The server must provide the processing capabilities to do this versus requiring clients to provide it. This simplifies the client which is the intent.

The syntax of the boolean query language is constrained to provide simplicity of query generation and translation without loss of useful capabilities. This simplification is accomplished by considering the user interface paradigm that drives the generation of boolean queries.

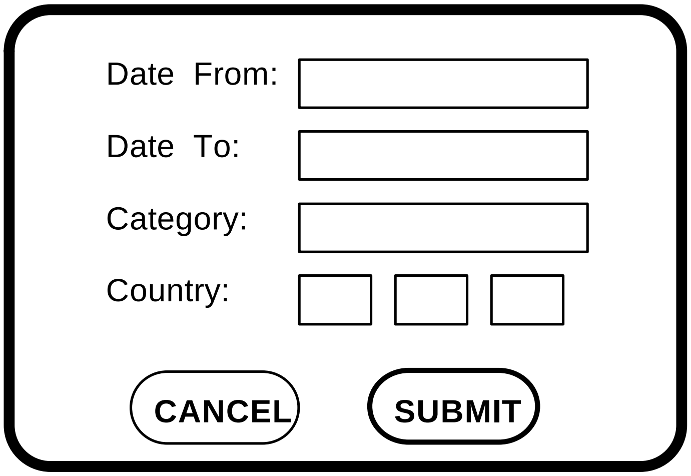

Figure 6-1 Example User Interface Query Form

In a user interface query form, as shown in Figure 6-1, there are a number of named fields which must be inputted by the end-user to generate a query. Each field implies a simple attribute relation, for example:

```
ATTRIBUTE > VALUE
```

This simple relationship is true of most fields, except those which accepted multiple values (such as the Country field shown above). For example, if the user fills in both US and UK for Country, indicating a search in both countries, then the boolean query would contain an "or" relationship:

```
(Country like 'US' or Country like 'UK')
```

In this example, the "like" operator is the equivalence operator for text expressions, which also supports wildcards (See E.2 below).

The logical relationship between fields can be an "and" relationship. For example if one specifies both Date From and a Country:

```
Date > '29-Feb-1996' and Country like 'US'
```

Taking all of the above into account as our query generation paradigm, one can constrain the boolean query syntax to a boolean product-of-sums, where most of the sums are just simple attribute relations. When it is not a simple relation, it is a logical sum expression, based upon the same attribute.

Formally, this can be expressed in the syntax of the Backus-Naur Form (BNF) specification as defined in Section 6.3 below.

6.3. BNF Rules

The Backus-Naur Form (BNF) for the boolean query syntax is show below. The following rules use the same BNF conventions as used in the OMG IDL technical reference. [5]

```
<boolean Query> ::= <Product Expression>*

<Product Expression> ::=
    <Sum Expression>  { "and" <Sum Expression> }*

<Sum Expression> ::= <Relation Expression>
 | "(" <Relation Expression> { "or" <Relation Expression> }+ ")"

<Relation Expression> ::=
    <Attribute>  <Relation Operator>  <Constant Expression>

<Relation Operator> ::=
    "=" | ">" | "<" | ">=" | "<=" | "<>" | "like" | "not like"
```

The BNF rules are augmented by the following constraints:

1. Constant expressions include the options defined in SQL92, except as otherwise noted here.
2. The <Attribute> contained in each relation within a sum expression are the same attribute. The operators are limited to "=", "<>", "like", and "not like" within sum expressions.
3. Wildcard expressions are allowed using the character "%" to denote a match with 0 or more characters.

For example:

```
attribute like 'target%'
```

would match the following strings:

```
'target'  "target9'  'target123'
```

The "like" and "not like" operators are the only operators used for text expressions and the only operators supporting wildcards.

Wildcards can be used to implement the effect of many character matching operations, such as: contains, begins with, ends with, not contains, not begins with, not ends with, and so forth.

For example:

```
attribute like '%contains_this%'
attribute like 'begins_with_this%'
attribute like '%ends_with_this'
attribute not like '%will_not_contain_this%'
attribute not like
'will_not_begin_with_this%'
attribute not like '%will_not_end_with_this’
```

6.4. BNF Semantics

A boolean Query is the starting token of the BNF definition of the query language. In other words, all allowable queries are generated from a boolean Query.

A Product Expression is a logical sum of products, i.e. a series of expressions that are ANDed together.

A Sum Expression is either a simple attribute relation or a series of simple relations ORed together. One can always detect the second case, because a parenthesis "(" occurs first.

A Relation Expression is a simple relationship between a particular attribute and a constant value. Constant values may be integers, floating point, strings, including the options allowed in the SQL92 standard.

6.5. Attribute Metadata

Interoperability will be limited if clients generate queries that require excessive processing. Therefore each implementation may identify specific attributes as queryable. In addition, the implementation may also define the allowable relation operators and if wildcards are allowed, as well as other characteristics.

This metadata information could be available as a set of parameters, retrievable through the Parameters interface.

7. Profile & Notification Facility

This section describes the semantics and sequencing of the Profile & Notification Facility. The intent of this facility is to allow clients to register their interests concerning geospatial information and to be notified when information relevant to their registered interests enters an archive.

This facility defines two interfaces, PN, the Profile & Notification interface itself and PNF_Callback, a set of callback interfaces. The PN interface is defined within the IAS module and the PNF_Callback interface is defined within the IA_CL module.

7.1. Inherited Methods

```
module IAS {


    // Profile & Notification Interface    “IAS::PN”
    interface PN: CA { // Inherits from Catalog Access
                            Interface
```

The Profile & Notification interface, PN, defined within the IAS module, inherits all of the operations from the IAS::CA interface, and defines several new operations.

The PNF server re-uses the methods inherited from the CAF with basically identical semantics. The four query methods (boolean_query, polygonal_query, elliptical_query and point_query) are used to submit standing queries and the get_results allows result sets from standing queries to be retrieved. The free_results method differs slightly. In the CAF, queries and their associated results are considered transitory. Freeing a transitory result deletes all results. In contrast, in the PNF, queries and results are considered to be persistent . Freeing the results from a standing query, only deletes those results that have already been retrieved by the client. This also means that in the PNF, a QueryId returned by submitting a standing query is valid for an indefinite time. This is different from a QueryId returned from a CAF query, which is only valid during the session in which it was issued by the server.

7.1.1. Profile & Notification Interface

```
struct QueryStatus {
    QueryId query_id;
    boolean new_results;
    };
sequence sequence<QueryStatus> QueryStatusList;

exception BadEmailAddress {ExceptionInfo};

// Forward reference to the PNF_Callback interface
interface PNF_Callback;
```

The structure QueryStatus is contains a QueryId to identify the query and a boolean which indicates whether or not that query has any new results.

```
void list_queries( out QueryStatusList queries )
    context(  “ContextInfo” );
```

The list_queries operation returns a status list of all of the standing queries. The output argument is a QueryStatusList which contains one element for each standing query from this client. The query_id member uniquely identifies each query with respect to the requesting client. The new_results member indicates if there are new query results to be retrieved. That is, new_results equal to TRUE indicates that a query has results that have not been retrieved. For example, if a standing query has 100 results and the client retrieves 10 (using get_results) the new_results flag should still be set to TRUE, because 90 hits remain. This also means that repeatedly checking status (using list_queries) does **not** change that new_results status. The client is responsible for maintaining the correlation between the QueryId and the details of the query (i.e. query parameters, human-readable description of the query etc.). The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

```
void remove_query( in QueryId query_identifier )
raises( BadQueryId )
        context( “ContextInfo” );
```

The remove_query operation allows the cancellation of a standing query. The BadQueryId exception is returned from operations if there is no standing query with the indicated query_id. The standard exception BAD_INV_ORDER (routine invocations out of order) will be returned if the server has not been successfully opened prior to this method being used.

```
    void request_not ification (
        in QueryId query_identifier,
        in IA_CL:PNF_Callback callback_objectref,
        in string email_address)
    raises( BadQueryId,BadEmailAddress )
        context(  “ContextInfo” );
```

The request_notification method indicates that the client wishes to be automatically notified of new “hits” against a standing query. The client supplies a QueryId to indicate the standing query of interest. It also supplies an object reference for a PNF_Callback object for the server to notify. On receiving hits against this standing query, the PNF server will invoke the notify method on this object reference. (see PNF_Callback below). The client can also provide a string containing an email address. On receiving hits against a standing query, an email message describing those hits will be sent to that address. Either the object reference or the address parameter may be NULL, indicating that client does not wish to be notified by that mode, but both cannot be NULL, The standard exception BAD_PARAM would be raised if both are NULL. The BadEmailAddress exception would be raised if an invalid email address is supplied. The exception BadQueryId is returned if the QueryId is invalid. The standard exception INV_OBJREF (invalid object reference) is returned if the client submits an invalid object reference for the PNF_Callback.

```
void request_push (
        in QueryId query_identifier,
        in IA_CL:PNF_Callback callback_objectref )
raises( BadQueryId )
context(  “ContextInfo” );
```

The request_push operation indicates that the client wishes to be sent all images resulting from hits on the standing query identified by query_identifier. The client supplies a QueryId to indicate the standing query of interest . It also supplies an object reference for a PNF_Callback object for the server to contact prior to sending the imagery. When the a PNF server has imagery to be pushed to the client it will invoke the push method on this object reference. (see PNF_Callback below). The exception BadQueryId is returned if the QueryId is invalid. The standard exception INV_OBJREF (invalid object reference) is returned if the client submits an invalid object reference for the PNF_Callback.

7.2. PNF_Callback Interface

The PNF_Callback interface is implemented by any **client** that wishes to be automatically notified of standing query hits or to have images automatically pushed to it**.** Clients that do not implement this interface cannot use the request_push method or the callback mode of the request_notification method. They may use the email mode of request_notification.

```
module IA_CL {

interface PNF_Callback {
```

The PNF_Callback interface, contained in module IA_CL, defines two methods which a PNF server can invoke to either notify the client of “hits” against a standing query or request a location to place an incoming “pushed” image.

```
void notify(
    in IA:QueryId query_identifier,
    in IA:QueryResults results)
raises (BadQueryId);
```

The notify method is invoked by a PNF server to notify the client that a standing query has new hits. The server will supply the QueryId of the standing query in the first parameter query_identifier and pass the details of the hits in the second parameter results.

```
InfoSandR::LocationSpecList push(
                       in IA:QueryId query_identifier,
                       in IA:QueryResults results)
            raises( BadQueryId);
```

The push method is invoked by a PNF server when it has imagery to be pushed to the client. The standing query that the imagery resulted from is passed in the first parameter query_identifier and the details of the incoming imagery are passed in the second parameter results. The client is required to return a LocationSpecList containing locations for the incoming imagery in the same order as the results, that is the first LocationSpec in the LocationSPecList corresponds to the first result in the QueryResults etc. For any images that are not wanted, a NULL or empty LocationSpec is returned.

Bibliography

1. *USIS Standards and Guidelines -- USIS Standards, Guidelines, and Conventions,* Section 4, Central Imagery Office, Vienna, VA, May 1995.

2. *National Imagery Transmission Format (NITF),* Version 2.0, U.S. DoD MIL-STD 2500A.

3. *Interface Control Document for IPA 1.0*, GTE Government Systems Corporation, Document Number 1947004D, February, 1995.

4. *System III Application Programmers Interface*, CDR1 Update, Document Number 60770-6043-SX-00, July 1994.

5. *CORBA: The Common Object Request Broker Architecture and Specification*, Revision 2.0, Object Management Group, Framingham, MA, OMG Document Number 93.12.43, December, 1993.

6. *CARS Mission Intelligence Segment: Trusted Image Storage Manager* (Briefing), Loral Western Development Laboratories, San Jose, CA, June 1995.

7. Mowbray, T.J., and Zahavi, R., The Essential CORBA: System Integration Using Distributed Objects, John Wiley & Sons, New York, 1995.

8. *DISCUS Programmer's Reference Manual, Developer's Guide, and Release Notes,* The MITRE Corporation, McLean, VA, May 1995.

9. *ImNet with CORBA Extensions* (Briefing), The MITRE Corporation in cooperation with TASC, McLean, VA, June 1995.

10. *USIS Image Access Business Object Modeling* (Briefing), USIS Architecture Team, Vienna, VA, June 1995.

11. *USIS Image Access Standards Analysis* (Briefing), USIS Architecture Team, Vienna, VA, May 1995.

12. *Addendum: Summary of Migration Systems Analysis,* USIS Architecture Team, Vienna, VA, June 1995.

13. *United States Imagery System Master Glossary* , Version 2, Central Imagery Office, Vienna, VA, August 1994.

14. *CORBAfacilities: The Common Facilities Architecture,* Version 4.0, Object Management Group, Framingham, MA, November 1995.

15. *CORBAservices: Common Object Services Specification,* Revised Edition, Object Management Group, Framingham, MA, March 1995.

16. *Object Query Service Specification: Joint Submission,* Document 95.1.1, Object Management Group, Framingham, MA, January 1995.

17. *Common Imagery Interoperability Facilities,* Central Imagery Office, Vienna, VA, March 1996.

18. *United States Imagery System: Standards Profile for Image Archives,* Document ASD SIA 05940000, Central Imagery Office, Vienna, VA, July 1994.

19. International Standards Organization, *Image Processing and Interchange: Programmer's Imaging Kernel (IPI-PIKS),* ISO/IEC IS 12087-1:1993.

20. *Joint Requirements Document for the USIS 2000 Accelerated Architecture Acquisition Initiative (A3I),* Central Imagery Office and the IMINT Directorate, Version 1.0, December 1996.

21. Common Imagery Interoperability Profile for Imagery Access, Central Imagery Office, Draft, Vienna, VA, June 1996.

22. Accelerated Architecture Acquisition Initiative (A3I) Requirements Document, Rev. 1.0, CIO-2054.

23. Joint Requirements Document for the USIS 2000 Accelerated Architecture Acquisition Initiative (A3I), Central Imagery Office and Rome Labs, Version 1.0, June 1996.

Appendix A: *Storage and Retrieval Facility IDL*

```
module InfoSandR {

#define ExceptionInfo any exception_info;

// PRODUCT REQUEST TYPE DEFINITIONS
    interface Product {};
    typedef any LocationSpec;
    typedef sequence<LocationSpec> Locat  ionSpecList;
    exception BadProductReference { ExceptionInfo };
    exception BadLocationSpec { ExceptionInfo };
    typedef string RequestId;
    typedef sequence<RequestId> RequestIdList;
    struct NameValue { string name; any value; };
    typedef sequence<NameValue> NameValueList;
    typedef sequence<string> NameList;
    exception BadName { ExceptionInfo };
    exception BadValue { ExceptionInfo };
    enum ResponseService { IMMEDIATE, QUEUED };
    exception ResponseServiceNotAvailable { ExceptionInfo };
    exception TooManyRequest  s { ExceptionInfo };
    exception ProductUnavailable {ExceptionInfo };
    enum CompletionState
        { COMPLETED, IN_PROGRESS, ABORTED,   CANCELED ,
           PENDING, OTHER };
    exception BadRequestId { ExceptionInfo };
    exception BadCreationAttributes { ExceptionInfo };

// PRODUCT REQUEST INTERFACE
    interface ProductRequest {

        // Request to transfer whole products
        RequestIdList disseminate(
            in Product product_to_disseminate,
            in LocationSpecList destinations,
            in ResponseService service )
            raises ( BadPr oductReference, BadLocationSpec,
            ResponseServiceNotAvailable, TooManyRequests )
            context(  “ContextInfo” );

        // Check completion status of request
        CompletionState check_completion(
            in RequestId request_identifier,
            out string state_information )
            raises ( BadRequestId )
            context(  “ContextInfo” );

        // Cancel outstanding request
        CompletionState cancel(
            in RequestId request_identifier )
            raises ( BadRequestId )
            context(  “ContextInfo” );

        // Store client generated information    product
        Product create( in LocationSpec initial_product_data,
            in NameValueList creation_attributes,
            out RequestId request_id)
            raises (BadCreationAttributes, BadLocationSpec )
            context(  “ContextInfo” );

    };

// ARRAY Type Definitions

    // Array is a special subtype of Product
    interface Array: Product  {};

    enum ElementType { BITDATA, BYTEDATA, SBYTEDATA, INT2DATA,
        SINT2DATA, INT4DATA,  SINT4DATA,  FLOAT4DATA,
        COMPLEXDATA, FLOAT8DATA, OTHERDATA };

    union Buffer switch (ElementType)   {
        case BITDATA: sequence<octet> bit_data;
        case BYTEDATA: sequence<octet> ubyte_data;
        case SBYTEDATA: sequence<char> byte_data;
        case INT2DATA: sequence<unsigned short> ushort_data;
        case SINT2DATA: sequence<short> short_data;
        case INT4DATA: sequence<unsigned long> ulong_data;
        case SINT4DATA: sequence<long> long_data;
        case FLOAT4DATA: sequence<float> float_data;
        case COMPLEXDATA: sequence<float> complex_data;
        case FLOAT8DATA: sequence<double> double_data;
        default: sequence<octet> other_d  ata; };

        typedef any RegionSpec;
        typedef sequence<RegionSpec> RegionSpecList;

        struct RegionData {
            RegionSpec region_spec;
            NameValueList region_header;
            ElementType element_type;
            Buffer region_data;
            };
        typedef sequence<RegionData> RegionDataList;
        exception BadRegionSpec      { ExceptionInfo };
        exception BadRegionData { ExceptionInfo };
        exception ArrayNotOpen { ExceptionInfo };

// ARRAY REQUEST INTERFACE
    interface ArrayRequest {

        // Prepare an array for region access
        void open _array(in Array product, in any access_kind)
            raises ( BadProductReference )
            context(  “ContextInfo” );

        // Deallocate an array  ’s resources - discontinue access
        void close_array(in Array product)
            raises ( BadProductReference, ArrayNotOpen )
            context(  “ContextInfo” );

        // Retrieve region data to memory
        void get_region(
            in Array product,
            in RegionSpec region_spec,
            inout RegionData region_data )
            raises( BadProductReference, BadRegionSpec,
                BadRegionData, ArrayNotOpen)
            context( “ContextInfo” );

        // Retrieve multiple regions to memory
        void get_multiple_regions(
            in Array product,
            in RegionSpecList region_specs,
            inout RegionDataList region_data_list )
            raises( BadProductReference, BadRegionSpec,
                BadRegionData, ArrayNotOpen )
            context(  “ContextInfo” );


    };
};
```

Appendix B:
Image Access Facility IDL

```
#include <InfoSandR.idl>

module IAS {

typedef string ClientContext;

// The data type supplied by convention for exception_info
struct ImageExceptionInfo {
    short status_code;
    string status_text;
    string exception_type;
    };
exception AlreadyConnected { ExceptionInfo };
exception BadOpenCriteria  { ExceptionInfo };
exception NoConnectionEstablished { ExceptionInfo };

interface Server {

        // Initialize library server connection
        ClientContext open(in NameValueList open_criteria)
            raises ( AlreadyConnected  , BadOpenCriteria)
            context(  “ContextInfo” );

        // Disconnection from library server
        void close()
            raises ( NoConnectionEstablished )
            context(  “ContextInfo” );
    };

exception CannotSet { ExceptionInfo };

interface Parameters {

    void get_parameters(
            in InfoSandR::NameList
names_of_parameters_requested,
            out InfoSandR::NameValueList parameter_values)
        raises( InfoSandR::BadName )
        context(  “ContextInfo” );

    void set_parameters(
            in InfoSandR::NameValueList parameter_values)
        raises( InfoSandR::BadName, InfoSandR::BadValue,
            CannotSet )
        context(  “ContextInfo” );
    };

interface ImageProduct: InfoSandR::Product, Parameters {};
interface ImageArray: InfoSandR::Array, ImageProduct {};

// Geographic Data Types
struct GeoCoords {
    double lat, lon; // degrees
    };
typedef sequence<GeoCoords> GeoCoordsList;
exception BadCoord { ExceptionInfo };


    // Exception Types
    const string CompressionNotAvailable
        = "Compression Not Available";
    const string FormatNotAvailable
        = "Format Not Available";

    enum ImageLocationKind {PathKind, HyperlinkKind,
        AddressKind};
    struct PathInfo {
        string user_name;
        string pass_word;
        string host_name;
        string path_name;
        string file_name;
        };

    // The Data Type supplied for Location Spec
    union ImageLocationSpec switch (ImageLocationKind) {
        case PathKind:
            PathInfo path;
        case HyperlinkKind:
            string hyperlink;
        case AddressKind:
            any address;
        };

    // Data Types Used with set and get parameters
    struct RegionParameters {
        unsigned long horizontal_size;
        unsigned long vertical_size;
        unsigned long resolution_level;
        };

    // Data type supplied for the get_region RegionSpe   c
    struct DisplayRegionSpec {
        long x_region_center;
        long y_region_center;
        };

    // Data type for the get_multiple_regions RegionSpec
    struct TileRegionSpec {
        long x_region_center;
        long y_region_center;
        unsigned long horizontal_size;
        unsigned long vertical_size;
        unsigned long resolution_level;
        };

// Image Access Interface “IAS::IA”
interface IA: Server, Parameters,
    InfoSandR::ProductRequest, InfoSandR::ArrayRequest {

    // Retrieve subimage covering geographic area
    InfoSandR::RequestId get  _subimage(
        in ImageArray image_array_product,
        in GeoCoords upper_left,
        in GeoCoords lower_right,
        in InfoSandR::LocationSpec location )
    raises( InfoSandR::BadProductReference, BadCoord,
        InfoSandR::BadLocationSpec, InfoSandR::TooManyRequests )
    context( “ContextInfo” );

  };

};
```

Appendix C:
Catalog Access Facility IDL

```
module IAS {

// Query Results
typedef string QueryId;
typedef sequence<string> AttributeValues;
  struct QueryHit {
        ImageProduct product_ref;
        AttributeValues att ributes;
        InfoSandR::RegionData browse_image;
        };
  typedef sequence<QueryHit> QueryHitList;
  struct QueryResults {
        InfoSandR::NameList attribute_names;
        QueryHitList query_hits;
        };

// Exceptions
exception BadQuerySyntax { ExceptionInfo };
exception BadAttribute { ExceptionInfo };
exception BadQueryValue { ExceptionInfo };
exception BadEllipse { ExceptionInfo };
exception BadQueryId { ExceptionInfo };
exception TooFewVertices { ExceptionInfo };
exception TooManyVertices { ExceptionInfo };

// Catalog Access Interface “IAS::CA”
interface CA: Server, Parameters {

    QueryId boolean_query(
        in string boolean_query_expression )
    raises ( BadQuerySyntax, BadAttribute,
                BadQueryValue, InfoSandR::TooManyRequests )
    context(  “ContextInfo” );

    QueryId polygonal_query(
        in string boolean_query_expression,
        in GeoCoordsList polygon_vertices)
    raises ( BadQuerySyntax, BadAttribute,
                BadQueryValue, BadCoord,
                TooFewVertices, TooManyVertices,
            InfoSandR::TooManyRequests )
    context(  “ContextInfo” );
    QueryId elliptical_query(
        in string boolean_query_expression,
        in GeoCoords ellipse_center,
        in double major_axis, // meters
        in double minor_axis, // meters
        in double azimuth,  // decimal degrees from North
        )
    raises ( BadQuerySyntax, BadAttribute,
                BadQueryValue, BadCoord, BadEllipse,
                InfoSandR::TooManyRequests )
    context(  “ContextInfo” );

    QueryId point_query(
        in string boolean_query_expression,
        in GeoCoords point_geo_location )
    raises ( BadQuerySyntax, BadAttribute,
                BadQueryValue, BadCoord,
            InfoSandR::TooManyRequests )
    context(  “ContextInfo” );

    void get_ results(
        in QueryId query_result_identifier,
        in long number_of_hits_to_return,
        out long number_of_hits_remaining,
        out QueryResults product_records )
    raises ( BadQueryId )
    context(  “ContextInfo” );

    void free_results(
        in QueryId query_result_identifier )
    raises( BadQueryId )
    context(  “ContextInfo” );

  };
};
```

Appendix D: Profile & Notification Facility IDL

```
module IAS {

struct QueryStatus {
    QueryId query_id;
    boolean new_results;
    };
sequence sequence<QueryStatus> QueryStatusList;

exception BadEmailAddress {ExceptionInfo};

// Forward reference to the PNF_Callback interface
interface PNF_Callback;

// Profile & Notification Interface “IAS::PN”
interface PN: CA { // Inherits from Catalog Access Interface

void list_queries( out QueryStatusList queries )
    context(  “ContextInfo” );

void remove_query( in QueryId query_identifier )
    raises( BadQueryId )
    context(  “ContextInfo” );

void request_notification (
        in QueryId query_identifier,
        in IA_CL:PNF_Callback callback_objectref,
        in string email_address)
    raises( BadQueryId,BadEmailAddress )
    context(  “ContextInfo” );

void request_push (
            in QueryId query_identifier,
            in IA_CL:PNF_Callback callback_objectref )
raises( BadQueryId )
context( “ContextInfo” );

    };
};


module IA_CL {
```

```
interface PNF_Callback {

void notify(
    in IA:QueryId query_identifier,
    in IA:QueryResults results)
raises (BadQueryId)
context( “ContextInfo”);

InfoSandR::LocationSpecList push(
            in IA:QueryId query_identifier,
            in IA:QueryResults results)
raises( BadQueryId)
context( “ContextInfo”);

    };
};
```

Appendix E:
Reference OMG Standard IDL

CORBA Standard Exceptions

```
#define ex_body {unsigned long minor; completion_status
completed;}

enum completion_status {COMPLETED_YES, COMPLETED_NO,
COMPLETED_MAYBE};
enum exception_type {NO_EXCEPTION, USER_EXCEPTION,
SYSTEM_EXCEPTION};

exception UNKNOWN ex_body;
exception BAD_PARAM ex_body;
exception NO_MEMORY ex_body;
exception IMP_LIMIT ex_body;
exception COMM_FAILURE ex_body;
exception INV_OBJREF ex_body;
exception NO_PERMISSION ex_body;
exception INTERNAL ex_body;
exception MARSHAL ex_body;
exception INITIALIZE ex_body;
exception NO_IMPLEMENT ex_body;
exception BAD_TYPECODE ex_body;
exception BAD_OPERATION ex_body;
exception NO_RESOURCES ex_body;
exception NO_RESPONSE ex_body;
exception PERSIST_STORE ex_body;
exception BAD_INV_ORDER ex_body;
exception TRANSIENT ex_body;
exception FREE_MEM ex_body;
exception INV_IDENT ex_body;
exception INV_FLAG ex_body;
exception INTF_REPOS ex_body;
exception BAD_CONTEXT ex_body;
exception OBJ_ADAPTER ex_body;
exception DATA_CONVERSION ex_body;
exception OBJECT_NOT_EXIST ex_body;
```

Appendix F: Related Facilities

The Mensuration Facility

The IAF and the Mensuration facility are related facilities that have distinct roles. These two facilities can be used independently or together to perform image access and mensuration.

When file-based retrieval is used, the Mensuration facility is used separately after image product transfer and retrieval of any attribute information from the image header and CAF.

When array-based retrieval is used, the IAF works in pixel space and the Mensuration facility works to transform between pixel space and other spaces, such as geographic coordinates.

An API is provided in IAF to support the transformation from client pixel space to source image space at the base (i.e., R0) level of the source image.

The Image Security Facility

The IAF and CAF do not attempt to address security issues in any comprehensive manner. The Image Security Facility is a future CIIF facility specification which has this architectural charter.

Minimal hooks are provided in IAF/CAF to make any subsequent changes to client and service code minimal due to the introduction of the Image Security facility.

The Locator Service

The Locator Service, defined in the CIIP, is a multi-faceted capability for managing client access to multiple image libraries. Part of the capabilities of the locator service are transparent to the client; relating to the automatic routing of retrieval requests to alternate libraries. These capabilities are part of the library implementation and are not within the scope of an interface specification.

Another locator service capability addresses client selection of libraries. In this case, there is system metadata information exposed to the client, and this would expose additional client interfaces. There is a commercial standard that can provide this service, i.e. the Trader Service.

The Trader Service originated with the Open Distributed Processing standards work at ISO. This work resulted in the fast track adoption of OMG IDL as the way to define software interface bindings for formal standards. The work continued at the OMG on the creation of a commercial API for the Trader Service (aka the Trader). This work is still in progress and is expected to be completed in 1996.

The Trader Service is a yellow pages directory service. Service offerers can advertise their capabilities in the Trader Services to enable discovery by clients. In their advertisements, service offerers include their IDL interface type and various characteristics that define and discriminate their services.

The Trader Service is highly applicable to the needs of the imagery Locator Service. Using the Trader Service, image libraries can advertise their service location, imagery coverages, and other characteristics. Clients can select the appropriate image library (IPL, CIL, NIL , etc.) based upon these characteristics. Each library can be uniquely referenced, using an object reference obtained from the Trader Service. The client can select the appropriate library based upon the choice of library object reference when using the Image Access Service APIs.

Glossary

This glossary contains a useful set of definitions for unique concepts in the Image Access Services Specification. Other standard terminology is defined in the USIS Master Glossary [13]. The terms defined in this glossary take precedence over other terminology definitions with respect to the contents of this specification.

Application Program Interface (API) - A high level language software interface, supporting high-level languages such as C, C++, Ada, and others. This expands upon the definition in the USIS Master Glossary [13].

Client - Any application software that accesses the image access services. This includes applications that search for image products, obtain image product attributes, and retrieves items from image libraries.

Server - Any software implementing one or more interfaces of the Image Access Facility or Catalog Access Facility. An IAF server stores image products and allows library clients to search its contents and retrieve image products. A CAF server stores metadata about image products, that enables discovery and retrieval of associated attributes.

Array- An Array is an image product that stores pixels or other array data (such as digital video or audio). Parts of arrays can be retrieved using the ArrayRequest interface of the Image Access Facility.

IDL Interface - A language independent API. IDL is defined in the CORBA standard [5] but can also be used independent of commercial object request brokers (ORB) to define APIs. The layered library system model shown in Figure G-1 shows the relationship between the IDL interface and the socket-level ICD interfaces.

Figure G-1 Interface layers within a typical legacy system

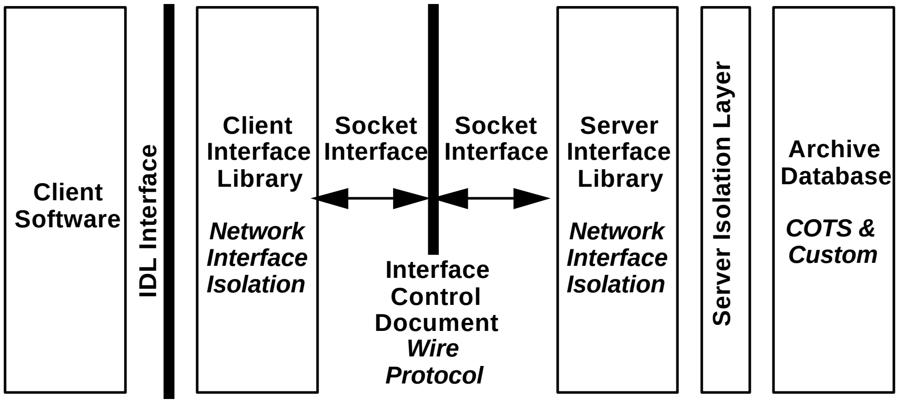

Image Processing and Interchange (IPI) - An international standard that defines image processing terminology, image processing operations, and interoperability specifications.

Image Product - Includes derived imagery products [13] (including text, graphics, pictures, database entries, voice reports, etc. see [13]) and in this definition, image products also include original source imagery and image chips.

Quality of Service- A general concept for variations in service provided by particular implementations of an API which do not have a direct effect upon the functional parameters in an operation. A quality of service is used directly in the facility to indicate whether a product transfer is immediate or queued.

Reduced Resolution Data Set (RRDS) - A subsampled version of a source image. Typically, RRDS are binary (power of 2) reductions of the source image.

Region - An area of an array product containing array data (or pixels if this is an image array product).

Subimage - A whole product that is created on the fly as the result of a get_subimage request. A subimage does not have to appear in the catalog.

Whole Product - A whole product is a self contained set of information that would typically reside in a disk file. An example of a whole product is any file-based image product or image. Whole products are distinguished from array regions and tiles which are partial products.

Acronyms

AIMS	Array Information Management System
API	Application Program Interface
BQS	boolean Query Syntax
CAF	Catalog Access Facility
CDR	Critical Design Review
CIIF	Common Imagery Interoperability Facilities
CIIP	Common Imagery Interoperability Profile
CIL	Command Image Library
CIO	Central Imagery Office
CORBA	Common Object Request Broker Architecture
COTS	Commercial off-the-shelf
ELT	Electronic Light Table
FTP	File Transfer Protocol
GIF	Graphics Interchange Format
GOTS	Government off-the-shelf
HTTP	Hypertext Transfer Protocol
IAF	Image Access Facility
IAS	Image Access Services
IDF	Imagery Dissemination Facility
IDL	Interface Definition Language
IPA	Image Product Archive
IPL	Image Product Library
ISO	International Standard Organization
NIL	National Image Library
OMG	Object Management Group
PNF	Profile and Notification Facility
RRDS	Reduced Resolution Data Set
S&R	Storage and Retrieval
SDE	Support Data Extension
TBD	To Be Determined
USIS	United States Imagery System

Points of Contact

Common Imagery Interoperability Working Group
Ron Burns, National Imagery and Mapping Agency
Phone: (703) 808-0891
Email: BurnsR@nima.mil

Project Lead - Support to CIIF Definition and Testing
John Polger, National Imagery and Mapping Agency
Phone: (202) 863-3004
FAX: (202) 488-0271

A3I CIIF Interface Definition
Charlie Green, Sierra Concepts, Inc.
Phone: (610) 347-0602
FAX: (610) 347-0602
Email: cpg@interramp.com

RFCs on this Specification
Tom Mowbray, PhD, The MITRE Corporation
Phone: (703) 883-6759
FAX: (703) 883-3315
Email: mowbray@mitre.org

Questions about this Specification & Support
Dave Lutz, The MITRE Corporation
Phone: (703) 883-7848
FAX: (703) 883-3315
Email: dlutz@mitre.org